A Study on Balancing Parallelism, Data Locality, and Recomputation in Existing PDE Solvers

C. Olschanowsky, M. Strout, S. Guzik, J. Loffeld, J. Hittinger

April 17, 2014

Super Computing
Denver, CO, United States
November 17, 2013 through November 22, 2013

A Study on Balancing Parallelism, Data Locality, and Recomputation in Existing PDE Solvers

Catherine Olschanowsky
and Michelle Mills Strout
Computer Science
Colorado State University

Stephen Guzik
Mechanical Engineering
Colorado State University

John Loffeld
and Jeffrey Hittinger
Center for Applied Scientific Computing
Lawrence Livermore National Laboratory

***Abstract*—Large-scale, PDE-based scientific applications are commonly parallelized across large compute resources using MPI. However, the compute power of the resource as a whole can only be utilized if each multicore node is fully utilized. Currently, many PDE solver frameworks parallelize over boxes, which ares rectangular domains of cells or faces in a structured grid. In the Chombo framework, the box sizes are typically 16^3 or 32^3, but larger box sizes such as 128^3 would result in less surface area and therefore less storage, copying, and/or ghost cells communication overhead. Unfortunately, typical on node parallel scaling involving shared memory parallelization over boxes or parallelization over cells within boxes performs quite poorly for these larger box sizes. In this paper, we investigate 30 different inter-loop optimization strategies and demonstrate the parallel scaling advantages of some of these variants on NUMA multicore nodes. Shifted, fused, and communication-avoiding variants for 128^3 boxes result in close to ideal parallel scaling and come close to matching the performance of 16^3 boxes on three different multicore systems for a benchmark that is a proxy for program idioms found in many Computational Fluid Dynamic (CFD) codes.**

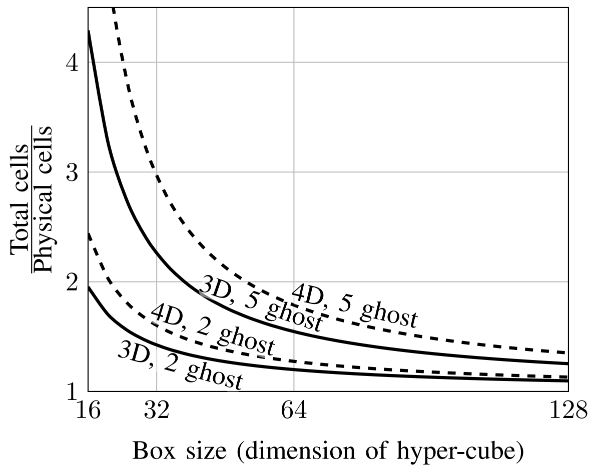


Fig. 1. Ratio of total cells to physical cells as a function of box size. A smaller ratio will result in the reduction of ghost cell overhead.

I. Introduction

Large-scale, structured-grid, PDE based scientific applications are commonly parallelized across nodes in large compute resources using MPI. Each MPI process operates on a set of boxes, where each box is a rectangular section of a structured grid. Since larger box sizes result in fewer ghost cells, the MPI parallelization prefers larger box sizes. Unfortunately, within a shared-memory, multicore node, straight-forward parallelizations involving parallelization over or within large boxes do not scale to the number of available cores. Since the number of cores per node grows in each new generation of machines, this limitation presents a significant issue for CFD codes in particular and all structured PDE codes in general. In this study, we present the advantages of applying inter-loop optimizations on structured CFD codes to improve on node parallel scaling for large boxes.

CFD codes consist of an outer time stepping loop, and, at each time step, various stencil computations are performed on the faces and cells within a structured grid. Two main performance bottlenecks are (1) the ghost cell updates that occur each time step and (2) the stencil computations.

The number of ghost cells is a function of the stencil width. A modern trend is to use at least a fourth-order algorithm (the error is $O(\Delta x^4)$ in (2)). Fourth-order and higher schemes have the advantage of increasing the solution accuracy per unit computer memory. However, a minimum of two ghost cells are required and up to five to accommodate non-linear stabilization mechanisms [32] and mapped grids [22]. For small boxes, the number of ghost cells can dominate the memory used. Figure 1 illustrates the ratio of total cells to physical cells for various problem dimensions and numbers of ghost cells. This is simply a plot of ratio $= \left(1 - \frac{2n_{\text{ghost}}}{n_{\text{cell}}}\right)^D$, where D is the number of space dimensions and n_{cell} is the number of cells in a single dimension of a box (assuming equal dimensions). A ratio of 1 means all the cells are physical cells. A ratio of 2 means that one has to exchange an amount of data equivalent to the physical solution domain to fill all the ghost cells. With 5 ghosts, a box size of 64 is necessary to get the ratio below 2. A box size of 128 is even more attractive to minimize memory wasted, especially in higher dimension (up to 6 can be used for kinetic calculations in phase space).

There are a number of optimizations such as communication hiding through communication and computation overlap [39] that can be used to reduce the cost of the ghost cell updates. In this work, we focus on the goal of moving to larger box sizes so as to reduce the cost of ghost cell updates no matter how the ghost cell updates are implemented.

In this paper, we focus on developing parallel implementations of stencil computations on large boxes that scale on multicore architectures. One option is to use MPI everywhere;

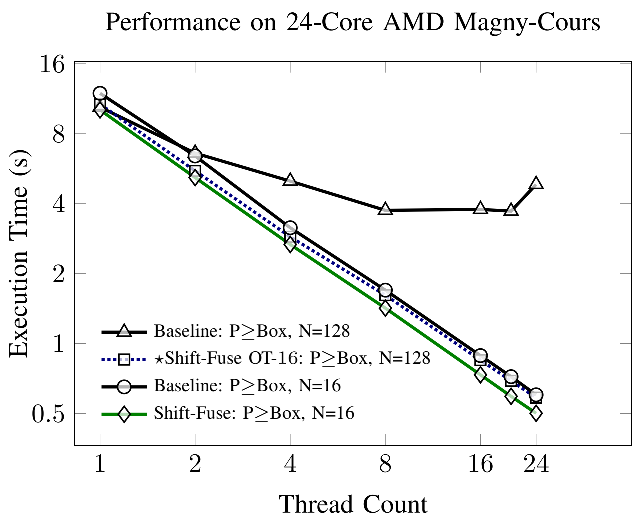


Fig. 2. On the Cray, the baseline, which is a parallelization over boxes ($P \geq Box$) results in poor on node scaling. A variant that performs loop shifting, fusion, and overlapped tiles of size 16^3 results in on node performance that matches that of the smallest box size. Additionally, the smallest box size of $N = 16$ scales just fine with a parallelization over boxes, but using a shifted and fused version of the computation another 16% improvement in the execution time is realized at 24 threads.

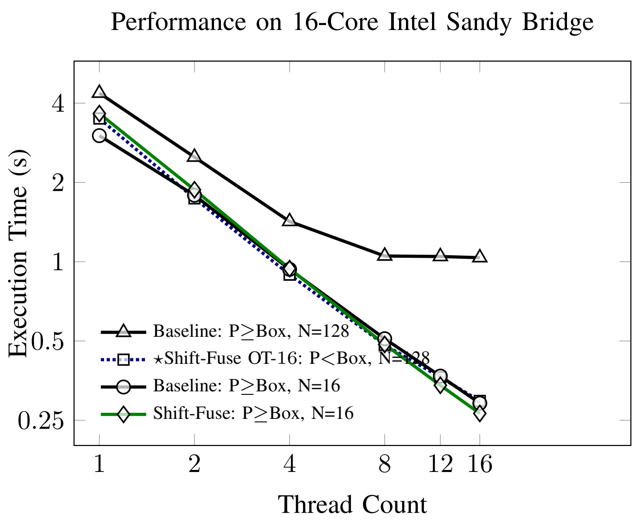


Fig. 4. On a Sandy Bridge, a shifted, fused, and overlapped tiling with tile size of 16^3 variant enables the $N = 128$ box size performance to match that of $N = 16$. Also a shifted and fused variant with no tiling improves on the $N = 16$ baseline.

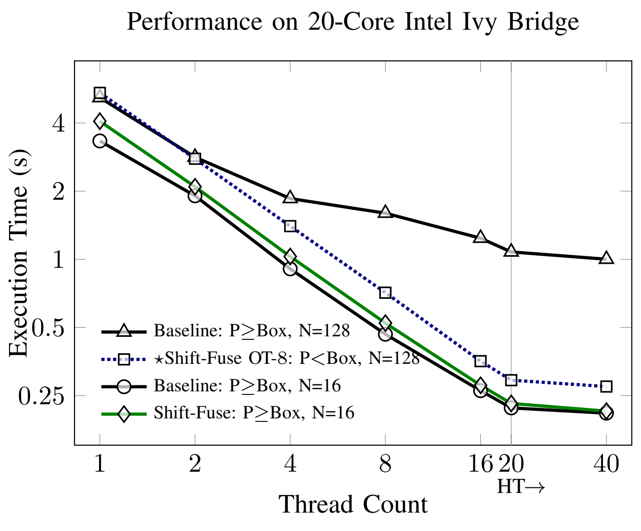


Fig. 3. On the Ivy Bridge, the $N = 128$ box size is still 2 times slower than the same amount of work parallelized with $N = 16$ box sizes, but a shifted, fused, and overlapped tiling with tile size of 8^3 variant does fix the parallel scaling.

a parallelization over boxes would then assign a set boxes per core on each node in the machine. As motivation, we approximate the impact of this approach on a multicore node using OpenMP and show results for box sizes of 16^3 and 128^3 in Figures 2, 3, and 4 (see the solid lines). The desired larger box size 128^3 exhibits poor parallel scaling within the multicore machines. The dashed line in Figures 2, 3, and 4 illustrate a shifted, fused, and overlapped inter-loop scheduling strategy that fixes the parallel scaling problem for the larger box sizes.

The primary contribution of this paper is to identify which of approximately 30 inter-loop parallelization scheduling variants results in this improved performance for three machines and to explain why. Although recent work has automated the application of certain inter-loop optimizations [50], [36], [5], the optimization strategies presented here are different, and the code complexity will make automating the approaches somewhat more difficult. Prototyping the strategies to determine their impact is an important first step in determining which strategies to automate.

Even prototyping these optimization strategies by hand within a benchmark code that uses an existing application framework is non-trivial. The CFD-motivated benchmark, which is detailed in Section III, is written using the Chombo [12] application framework. The well-established program idioms in Chombo result in clean, modular code that is both maintainable and displays excellent inter-node parallel scaling performance. However, these common program idioms do not ease the implementation of inter-loop optimization strategies that require the introduction of complex loop bounds, the careful management of temporary storage, and attention to details such as common subexpression elimination in array addressing arithmetic. To ease the process of developing so many variants for this study, we use CodeGen+ [8] to generate the complex loop bounds.

In Section II, we review some of the basics in CFD computations and abstractions such as the box that are used in application frameworks such as Chombo. In Sections III and IV, we present the CFD-motivated benchmark, the approximately 30 shared-memory variants of that benchmark, and how we are able to leverage a polyhedral code generation tool to implement the variants. In Section V, we describe how these variants differ from stencil computation optimizations in related work. In Section VI, we detail results from key categories of the variants. In Section VII, we conclude.

II. PDE Application Frameworks

Computational approaches are used to solve partial differential equations when the analytic solutions to them are not known. The partial differential equations often describe some conservation law in physics. For example, the Navier-Stokes equations describe conservation of mass, momentum, and energy in a fluid flow. These equations can generally be written in the form

$$\frac{\partial \mathbf{U}}{\partial t} + \boldsymbol{\nabla} \cdot \vec{\mathbf{F}}(\mathbf{U}) = 0\,, \tag{1}$$

where $\mathbf{U}$ is a vector of unknowns (.e.g, density, momentum, and energy), $\vec{\mathbf{F}}(\mathbf{U})$ is the flux dyad (tensor with components for each direction), t is time, and $\boldsymbol{\nabla}$ is the differential operator in space. To solve (1) numerically, it is approximated by a sequence of algebraic equations that are solved at discrete locations defined by a grid covering the spatial domain.

If (1) is approximated by point-wise values on the grid, the method is described as *finite-difference*. For example, using Taylor-series expansions,

$$\left.\frac{\mathrm{d}f}{\mathrm{d}x}\right|_i = \frac{f_{i+1} - f_{i-1}}{2\Delta x} + O(\Delta x^2)\,, \tag{2}$$

where $f_i \equiv f(x_i)$ and i denotes a discrete location on the grid (in one dimension). The term $O(\Delta x^2)$ is the discretization (or truncation) error induced by approximating the continuous differential equation with a discrete algebraic form. It is a function of the spacing of the grid and, being raised to the power of two, we would expect a quadratic decrease in the discretization error as the grid spacing is reduced. This would be labeled a second-order method. Note also that (2) has a stencil of size 1: the quantity f is required from grid points $i \pm 1\Delta x$ to compute $\frac{\mathrm{d}f}{\mathrm{d}x}$ at i.

Equation (1) could alternatively be integrated over a small control volume, $V_{\boldsymbol{i}}$, defined by the grid, i.e.,

$$\frac{\partial}{\partial t}\int_{V_{\boldsymbol{i}}} \mathbf{U}\,\mathrm{d}\boldsymbol{x} + \int_{V_{\boldsymbol{i}}} \boldsymbol{\nabla}\cdot\vec{\mathbf{F}}\,\mathrm{d}\boldsymbol{x} = 0\,. \tag{3}$$

Applying the divergence theorem of Gauss results in

$$\frac{\partial}{\partial t}\int_{V_{\boldsymbol{i}}} \mathbf{U}\,\mathrm{d}\boldsymbol{x} + \int_{\partial V_{\boldsymbol{i}}} \vec{\mathbf{F}}\cdot\hat{n}\,\mathrm{d}\mathcal{S} = 0\,, \tag{4}$$

where the integral of $\boldsymbol{\nabla}\cdot\vec{\mathbf{F}}$ over the control volume is converted into an integral of the normal component of $\vec{\mathbf{F}}$ over the surface of the control volume ($\hat{n}$ is a unit normal pointing outwards from the volume). The physical description of (4) is that the change of $\mathbf{U}$ in time equals the net balance of $\vec{\mathbf{F}}$ passing through the surfaces during that time. For example, the change in density equals the mass flux in or out of the control volume. Discretizations based on (4) are labeled *finite-volume* methods. Discretization involves finding approximations to $\vec{\mathbf{F}}$ on the faces of the control volumes based on the solution in nearby cells as shown in Figure 5.

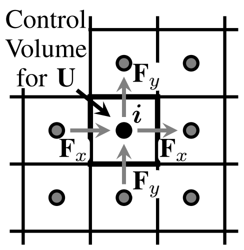


Fig. 5. Cell values affect flux across faces and vice versa.

As with finite-difference methods, the discretization imposes a truncation error and implies a stencil width. An advantage of finite-volume methods is a local conservation property that ensures discrete conservation over the entire domain, just as in the original partial differential equations.

Much of the work in solving PDEs then involves iterating over the cells in a grid and evaluating the algebraic equations created from the discretization. In structured grids, the solution content is stored in a multidimensional array. Large domains are partitioned into many smaller pieces so that the calculations can be performed in parallel. We call the partitions *boxes*. Layers of ghost cells are added around the boxes so that computations on the boxes can be performed independently. Before stencil computations begin, the ghost cells are filled with information from the physical cells sharing the same location. Outside the domain, boundary conditions may be used to set the ghost cells. The boxes are the coarsest grain of parallelism and are spread across nodes in a distributed environment.

Many application frameworks have been developed to ease the implementation of large-scale, complex, PDE-based simulation codes. We consider one classic varient of such frameworks: those based on structured, logically-rectangular meshes. Such meshes are typically used with finite difference and finite volume discretizations and have the advantage that data associativity is embedded implicitly within the regular multidimensional array access patterns. The fundamental building block is a logically rectangular patch or box. Examples include Chombo [12], SAMRAI [26], BoxLib [19], GrACE [44], Cactus [20], AMROC [15], AMRClaw [43], Unitah [31], and Overture [24].

Advanced structured grid frameworks encode many of the common operations associated with mesh and data management as well as support for solvers specific to the composite mesh structure. Examples include inter-patch interpolation routines, mesh refinement algorithms, and overset hole cutting algorithms. Support for at least distributed parallel data management and operations is also generally provided. Thus, the developer can re-use many PDE solution techniques based on serial, single-grid algorithms and can leverage utilities to construct the composite solution while letting the framework handle the parallel data management.

As an exemplar of this class of application frameworks, we consider the block-structured adaptive mesh refinement framework Chombo [12]. Like many of the examples [26], [44], [19], [15], [43], [31], Chombo supports the solution of steady-state and time-dependent PDEs based on finite difference and finite volume methods within the Berger-Oliger-Colella [6], [7] adaptive mesh refinement formulation.

Any time-dependent PDE simulation code has the same basic structure: initialize the mesh and solution, advance the solution in time (time step loop), and shut down. For structured grid frameworks, within each step of the time-advancement loop, calculations are done on each box, often with some communication and additional operations to make the solutions consistent across all of the boxes.

III. CFD Exemplar

In this work, we use a simple kernel representative of the stencil calculations performed on a box in CFD computations

```
 1 phi0 = phi1 = initial data
 2 for (every box)
 3 {
 4   for (int dir = 0; dir != SpaceDim; ++dir)
 5   {
 6     for (int iC = 0; iC != nComp; ++iC)
 7     {
 8       for (every face in direction dir)
 9         flux(face) = EvalFlux1(phi0);
10     }
11     velocity = flux[component dir+1];
12     for (int iC = 0; iC != nComp; ++iC)
13     {
14       for (every face in direction dir)
15         flux(face) = EvalFlux2(flux(face),
16                                velocity);
17       for (every cell)
18         phi1(cell) +=
19           flux(cell + 1) - flux(cell);
20     }
21   }
22 }
```

Fig. 6. Pseudo-code for finite-volume kernel exemplar. iC is a loop over components in **U** and SpaceDim = 3 is the dimension of the problem. Line 4 is a loop over the directions of the cell faces. The **for** statements tinted red on lines 8, 14, and 17 are nested loops over the spatial dimensions of the box.

to study the balance between parallelism, data locality, and re-computation in a node. The kernel is based on the finite volume method and includes a calculation of the flux on the faces of a cell and accumulation into the cell to update the solution.

A. Exemplar Pseudo-code

Pseudo-code for the benchmark is in Figure 6. This code fragment has a modular style since separate one-dimensional functions can be written for `EvalFlux1`, `EvalFlux2` and for the accumulation described by lines 18–19. The modular style makes it easy to test and the one-dimensional construction allow for changing `SpaceDim` at compile time. Additionally, it is highly parallel; at lines 8, 14, and 17, the granularity of parallelism is a single face or cell within the box. Figure 7 illustrates the computation as it occurs for a two-dimensional grid. All of the flux calculations in the X and Y directions, F_x and F_y, are computed using neighboring cell values. Cell values, or `phi0` and `phi1` in the pseudo-code, are indicated with white circles. The cell values are computed using the adjacent flux values.

However, there are two problems. First, the variable `flux` is a temporary. The size of the temporary can be reduced from covering the whole box, but only at the expense of parallelism (e.g., columns of cells in the x-directions could be evaluated without a temporary, but each column must be evaluated serially). Second, `phi1(cell)` is touched `SpaceDim` times, a problem that is difficult to overcome without introducing redundant calculations, large temporaries, or severely inhibiting parallelism. The difficulties arise because of the different iteration spaces for computing F_x, F_y, and for accumulating into cells (see Fig. 7).

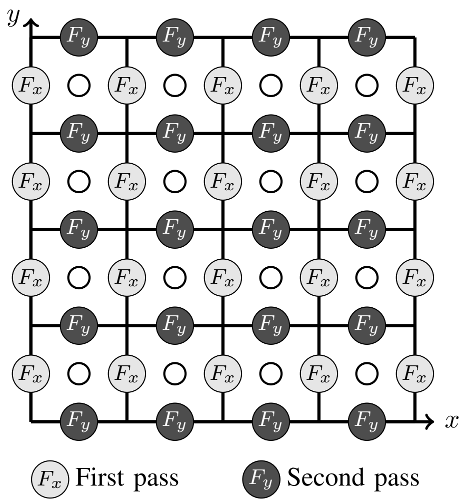


Fig. 7. Baseline iteration schedule with nested loops and separate passes for x-direction flux and y-direction flux and after each loop over fluxes in a particular direction the impact of that flux is accumulated into adjacent cells.

B. Exemplar Formulation

In the following, a simplified representation of a flux kernel is described; the variables and equations are not physically correct. In three spatial dimensions, the solution in the cells consists of cell average quantities of density, velocity, and energy

$$\langle \boldsymbol{\phi} \rangle_{\boldsymbol{i}} = \langle \mathbf{U} \rangle_{\boldsymbol{i}} = [\langle \rho \rangle \; \langle u \rangle \; \langle v \rangle \; \langle w \rangle \; \langle e \rangle]^T_{\boldsymbol{i}} \; . \tag{5}$$

The evaluation of the flux is performed in two parts. First, in `EvalFlux1`, the average solution on the face must be computed from the average solution in the cells [32].

$$\begin{aligned} \langle \boldsymbol{\phi} \rangle_{\boldsymbol{i}+\frac{1}{2}\boldsymbol{e}^d} = &\frac{7}{12} \left(\langle \boldsymbol{\phi} \rangle_{\boldsymbol{i}} + \langle \boldsymbol{\phi} \rangle_{\boldsymbol{i}+\boldsymbol{e}^d} \right) \\ &- \frac{1}{12} \left(\langle \boldsymbol{\phi} \rangle_{\boldsymbol{i}+2\boldsymbol{e}^d} + \langle \boldsymbol{\phi} \rangle_{\boldsymbol{i}-\boldsymbol{e}^d} \right) + O(\Delta x^4) \, . \end{aligned} \tag{6}$$

In (6), $\boldsymbol{e}^d$ is a unit vector in direction d, and $\boldsymbol{i} + \frac{1}{2}\boldsymbol{e}^d$ is therefore the high-side face in direction d from cell $\boldsymbol{i}$. Next, as part of `EvalFlux2`, the flux is computed as

$$\Delta x \langle \mathbf{F}_d \rangle_{\boldsymbol{i}+\frac{1}{2}\boldsymbol{e}^d} = \langle \boldsymbol{\phi} \rangle_{\boldsymbol{i}+\frac{1}{2}\boldsymbol{e}^d} \Big|_{d+1} \langle \boldsymbol{\phi} \rangle_{\boldsymbol{i}+\frac{1}{2}\boldsymbol{e}^d} \, , \tag{7}$$

where $|_{()}$ is a vector subscript counting from 0. E.g., for $d = 0$,

$$\Delta x \langle \boldsymbol{F}_0 \rangle_{\boldsymbol{i}+\frac{1}{2}\boldsymbol{e}^0} = \langle u \rangle_{\boldsymbol{i}+\frac{1}{2}\boldsymbol{e}^0} [\langle \rho \rangle \; \langle u \rangle \; \langle v \rangle \; \langle w \rangle \; \langle e \rangle]^T_{\boldsymbol{i}+\frac{1}{2}\boldsymbol{e}^0} \, , \tag{8}$$

Although simplified, the flux kernel provides a realistic ratio of arithmetic intensity. It also presents two challenges: (1) loops in the kernel have different centerings and cannot be easily fused and (2) a specific component from `EvalFlux1` is required for computing all components in `EvalFlux2`, thus incurring non-trivial dependencies.

C. Exemplar Implementation Details

The baseline implementation of the exemplar illustrated in Figure 6 is implemented using the Chombo. The current parallel strategy in Chombo is to use MPI everywhere; each core is assigned an MPI process. For load balancing purposes, hundreds of boxes can be assigned to each process.

The data associated with a box is represented by an `FArrayBox`, which contains a pointer to a float or double data array in column-major ordering. In the exemplar, the nested loops over the spatial dimensions of the box (red for loops in Fig. 6), are all written in C++. Our experience is that Fortran reliably and most easily yields the fasted code for stencil operations. However, we can reproduce the performance in C++, by caching pointer offsets for each cell in a stencil and using these offsets along with pointer arithmetic in each "unit-stride" column of cells in a box.

The data layout for $\mathbf{U}$ on a three-dimensional grid is $[x, y, z, c]$ where c is a component of $\mathbf{U}$ (see equation (5)) and Fortran ordering is assumed (x is unit stride). There are pros and cons to this layout, but it works well for gradient calculations. Nevertheless, for the flux kernels, it is somewhat disadvantageous because the components of velocity are required to compute each component of flux, and the individual components in a cell are very far apart in memory. The data layout cannot be changed unless one wishes to repack all the cell data for some segment of code.

The exemplar implementation is compiled for three dimensions and 64-bit floating point numbers. A single node is used for the calculations with OpenMP used for parallel calculations. The parallel granularity is tested for both boxes, and z-slices within a box. The total number of cells per solution is 5,0331,648 which may be divided into 12,288 boxes of size 16^3, 1,536 boxes of size 32^3, 192 boxes of size 64^3, or 24 boxes of size 128^3. The number of cells in the exemplar were selected to provide at least 1 box for each of 24 threads, but real applications contain significantly larger problem sizes.

IV. Inter-Loop Parallelization Variants

The hypothesis is that the poor on node parallel scaling of larger box sizes is due to a memory bandwidth bottleneck. Reducing the traffic between memory and each thread in the parallelization requires improving the data locality in the computation and reducing the amount of temporary storage as much as possible. The tradeoff is non-trivial because changing the schedule for a computation for improved data locality and minimal temporary storage often removes all possible parallelization opportunities. There is a known tradeoff between parallelism, data locality, and redundant computation.

The goal in this work is to investigate the parallelism, data locality, and redundant computation tradeoff in a Chombo CFD exemplar and using inter-loop schedules that have not been attempted before in such a context due to the implementation complexity involved. The shared memory parallel variants in this study fall under the following broad categories:

- Series of loops in the original exemplar
- Shifted and fused loops
- Shifted, fused, and tiled with wavefront parallelism
- Overlapped tiles (aka communication avoiding)

In this section, we detail each of the above categories and their impact on the parallelism, data locality, and temporary data as well as how it interacts with other axes in the space of variants. Table I summarizes the temporary data size for each category. Additionally, we describe how leveraging existing code generation tools reduces the implementation complexity.

TABLE I. A summary of the implementations: N is the number of elements in a single dimension (box size is N^3), T is the number of elements in a single dimension in a single tile, C is the number of components (5), P is the number of threads.

Schedule	Temporary Data
Series of Loops	Flux:$C(N+1)^3$ Velocity:$(N+1)^3$
Loops shifted and fused	Flux:$2+2N+2N^2$ Velocity:$3(N+1)^3$
Loops shifted, fused, tiled	Flux: $2(3CN^2)$ Velocity: $3(N+1)^3$
Shifted, fused, overlapping tiles	Flux:$PC(2+2T+2T^2)$ Velocity:$PC(3(T+1)^3)$

A. Original: Series of Loops

The original code corresponds to Figure 6. Figure 7 illustrates the schedule for this category of variants for a two-dimensional box. Execution of this schedule results in reading the input data and writing the output data three times, once for each dimension. Implementations in this category vary along the following axes: (1) parallelization over boxes or within boxes, and (2) the component loops at lines 6 and 12 outside as they are shown or inside the loops over faces and cells.

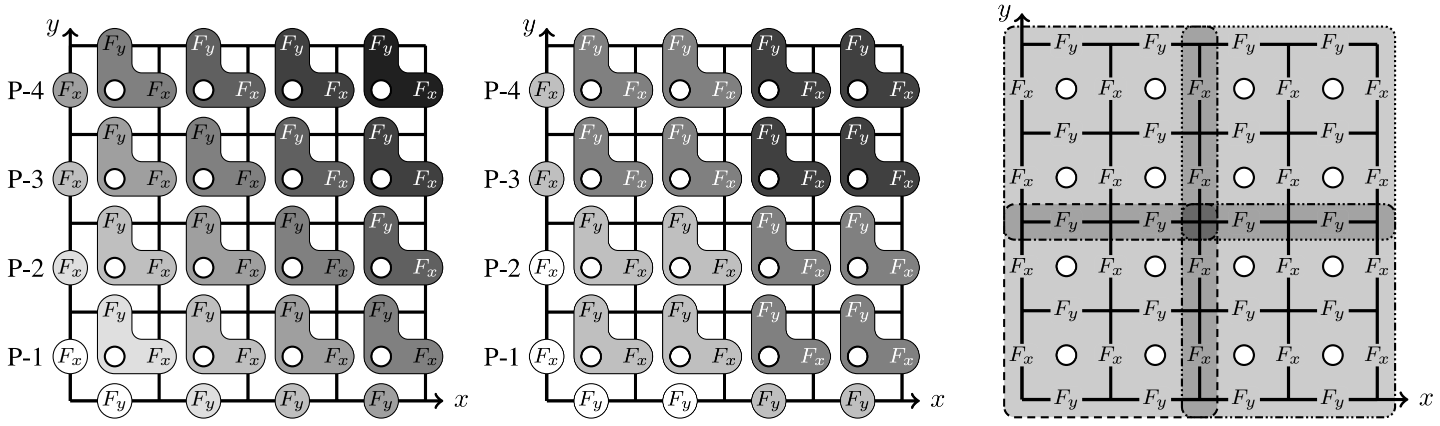
Temporary Data. These schedules requires $O((N+1)^3)$ temporary data for holding flux values between operations and $O((N+1)^3)$ for velocity data. Storage reuse occurs as the flux arrays are first written to during both flux steps and first read during the accumulation statement. In the "component loop on the outside variant" no temporary storage is required for the velocity data.

Temporal and Spatial Data Locality. The traversal order in this schedule yields a high degree of spatial locality in the X-direction. The spatial locality is lower for the Y and Z accesses of the stencil. The temporal locality depends on the size of the data and the size of the cache. For large problem sizes, it is possible for the input data and temporary data to fall out of cache before reuse, causing this schedule to have almost no temporal data locality.

Parallelism. Each of the face and cell loops in isolation is fully parallel. However, for small values of N (box size in each dimension), there is not enough work to justify parallelization within the box. Recall that each node has a number of boxes assigned to it. The parallelization, when done over boxes, is much more effective.

B. Shifted and Fused

The three loops within a box in the exemplar are not candidates for simple loop fusion. This is due to the structure being iterated over (faces versus cell centers) as well as stencil-shaped data dependencies. To overcome these limitations, the face loops can be shifted and then fused with the loop over cells. The component loops complicate this more, but Figure 8(a) illustrates the shift and fuse for a 2D box example when the component loops are on the inside. The concept



(a) To improve data locality for the pseudocode in 6, one can shift the face loops and fuse them with the cell loops. For parallelism the fused iterations can be executed in wavefronts, where all of the iterations in a wavefront can be executed concurrently.

(b) Wavefronts of blocked, or tiled, fused iterations.

(c) Overlapped tiles can all be executed in parallel. Flux computations on the surface of tiles are computed by each tile that uses them. This can be done with shifted and fused or the original schedule.

Fig. 8. Variant data access patterns on a 2D structured grid. The fluxes, F_x and F_y are computed on cell faces. The final result, `phi1`, is stored on the cell centers marked by ○

extends to the 3D boxes in the exemplar and component loops on the outside.

Implementations in this category vary along the following axes: (1) parallelization over boxes or within boxes, (2) component loops on the outside or inside, and wavefront parallelization at the per iteration granularity or not.

Temporary Data. The shifted and fused schedule requires less temporary storage overall and different amounts for the flux data in each direction. In the X-direction (innermost loop), the schedule calculates $flux_lo$, $flux_hi$ for each cell, and then immediately uses them in the same iteration in the accumulation. Therefore, in the X-direction only 2 scalars are needed. In the Y-direction the $flux_hi$ value used by cell (0,0,0) will be used as the $flux_lo$ value for cell (0,1,0). $O(N+1)$ flux values must be saved for the y-direction. The Z-direction requires one more dimension of data to be saved, essentially an entire plane of data $O(N+1)$. The velocity temporary data required for this schedule remains $O(3(n+1))^3$.

Temporal and Spatial Data Locality. The spatial locality with respect to each operation in this schedule remains the same as for the original schedule. The real difference is in the temporal locality with respect to the flux data. Multiple operations are performed for each face or cell during the same iteration and so the temporal locality is greatly increased. In the case of a perfect fuse it would be possible to read each piece of data in only once (for the x-direction), however, this fusion is not perfect and therefore some of the data still needs to be read twice.

Parallelism. Due to the fusion, there are now data dependencies between tiles. Parallelism can be partially recovered by utilizing a wavefront traversal. The wavefront traversal has the well-known problem that for the first and last wavefronts of iterations there will not be enough tasks to occupy all of the available cores. Additionally, wavefront parallelism at the granularity of single fused iterations ruins spatial locality in the X-direction.

C. *Tiling with Wavefront Parallelism*

This schedule involves running the set of tiles in a wavefront pattern as seen in Figure 8(b). Implementations in this category vary along the following axes: (1) parallelization over boxes or within boxes, (2) component loops on the outside or inside, and (3) tile sizes of 4, 8, 16, or 32.

Temporary Data. This execution schedule requires a codimension flux cache. In the variant that maintains the component loop as the outside loop, the flux cache for the exemplar application is 3D. In the variant that moves the component loop to the inside more space is required, a 4D cache, in order to save multiple component values simultaneously.

Temporal and Spatial Data Locality. Using cube tiles simultaneously reduces the spatial locality while increasing the temporal locality. This is because the streamed accesses in the X-direction are interrupted, however, the reuse distance between data in the Y and Z-direction stencil accesses is reduced.

Parallelism. The wavefront pattern suffers from the disadvantage that it lacks sufficient parallelism to occupy all of the cores on a multi-core node during the initial and finishing wavefronts.

D. *Overlapped Tiles*

It is possible to expand all of the tiles by one plane of flux operations in each direction and remove all data dependencies between tiles. This approach is shown in Figure 8(c). Overlapped tiles have the disadvantage that some computation is being repeated across tiles. However, there is a significant increase in parallelism over wavefront tiles, and data locality is maintained within each tile.

This approach is different than just maintaining the use of smaller tiles during on-node calculations due to ghost cells in a distributed parallelism model. The extra layer of cells computed by the overlapped tiles do not result in duplicated data, rather in data sharing. Therefore, the impact on performance is significantly less.

This schedule involves running the set of tiles in a wavefront pattern as seen in Figure 8(b). Implementations in this category vary along the following axes: (1) parallelization over boxes or within boxes, (2) series of loops schedule in each tile or shifted and fused loops within each tile, (3) component loops on the outside or inside, and (4) tile sizes of 4, 8, 16, or 32.

Temporary Data. This scheduling method results in a significant decrease in temporary data. Each thread is only required to save a single tile of temporary data. Flux values require only $O((T+1)^3)$ bytes, where T is the number of cells in one direction of the tiles.

Temporal and Spatial Data Locality. Spatial locality is not heavily affected by this scheduling technique; however, due to the reduction in working set size, temporal locality is greatly improved. Depending on the tile size, it may be possible to maintain the full working set in cache.

Parallelism. There are no data dependencies between tiles. Therefore, the limitation on parallelism comes from the number of available cores rather than the schedule and storage mapping.

E. Leveraging Code Generation Tools

There are a large variety of transformations possible and some of the transformations are configurable. A total of 328 variants are possible, but many of those variants are not practical. For example, tile sizes were only used for box sizes that were strictly larger. Additionally, other variants were left out because it was clear they would not perform as well. For example, overlapped tiles did not use the component loops on the inside because the untiled component loop on the inside variants were slower than the component loop on the outside variants.

Due to the large number of variants , we use a polyhedral model based code generator, Omega+ [9] to automate the generation of loop bounds. The schedules were implemented by taking the original Chombo code and separating the computation into three distinct concerns: *What* is being calculated, *When* and with what parallelization is each iteration executed, and *Where* are values stored in temporary arrays. The *What* is specified with statement macros and an integer tuple set defining the domain of iterations for each statement macro. The *Where* is implemented with storage mapping macros that map indexed values to storage locations. The code can appear single assignment even though storage is being reused. The *When* can be specified as a schedule mapping. We then manually place an OpenMP parallel for pragma outside certain loops.

V. Related Work

Expressing and optimizing structured, stencil computations is an active area of research. To our knowledge none of the previous optimizations for structured grids except those in [50], [5], and [36] have investigated scheduling across loops that share data but implement different stencils, which is the typical pattern in CFD codes. In this section, we compare the work in this paper to those most closely related and to other research that optimizes stencil computations.

A. Optimizations for Stencil Computations

Previous approaches that most resembles the presented here using inter-loop scheduling techniques for regular grids are the hierarchical overlapped tiling work [50], stencil transformation extensions for Chill [5], and the Halide project [36].

Zhou et al. [50] present a compiler algorithm for performing hierarchical overlapped tiling for a sequence of loops. Their approach could be used to automate the schedules investigated here. Their results on benchmarks show that overlapped tiling and hierarchical overlapped tiling perform well. In comparison, they experiment on a GPU, whereas our work focuses on the multicore scalability issue. Additionally, we work with a series of loops that model the computations performed by many PDE application frameworks including all of the complexity involved with handling multiple variables at each cell and face in the grid.

Basu et al. [5] extend the program transformation scripting tool Chill [29] with loop fusion, overlapped tiling, and a form of wavefront computation within each overlapped tile for a computation that includes a red-black Gauss-Seidel algorithm. Our use of CodeGen+, which is part of Chill, indicates that the set of loop transformations we study here could be incorporated into Chill.

Halide [36] provides a domain-specific, functional programming language to enable the specification of graphics computation pipelines, which contain stencil computations. The Halide compiler and autotuner determines the tradeoff between data locality, parallelism, and redundant computation that will result in the best performance for the given graphics pipeline on the give architecture. However, the inter-loop scheduling strategies that we present cannot be expressed in the scheduling language provided in Halide. Specifically, in Halide, it is only possible to shift and fuse two loops in one dimension, but we are shifting and fusing in three dimensions.

Optimizations that tile across a time loop containing a single stencil followed by parallelization are quite common. These approaches fall under two main categories: tiling with a wavefront parallelization or overlapped tiling.

The concept of tiling over time (or time skewing [47]) and then performing wavefront parallelization is similar to the shift, fuse, and wavefront schedules we perform except we are scheduling between different stencil loops instead of across time steps over the same stencil. 3.5D tiling [34] performs a blocked wavefront parallelism within a computation where the same stencil is applied many times within an outer time loop.

Overlapped tiling for stencil computations appears as early as 1997, when Bassetti et al. [4] introduced an optimization they call sliding block temporal tiling and when Sawdey and O'Keefe [40] developed a FORTRAN source-to-source translation tool that introduced overlapped tiles through time by creating a larger halo. The Cactus toolkit is an application framework that supports distributed memory parallelization of

PDE applications and includes some capability for doing overlapped tiles of computation to avoid communication [1]. Many other papers describe tools or experiments where overlapped tiles were used in regular stencil codes to avoid excess communication either between distributed processes or between non-shared caches in shared memory threads [16], [37], [27], [30], [21], [13], [33], [11], [10], [42], [25].

Avoiding redundant computation while avoiding the pipeline startup of wavefront parallelization has also been addressed in the diamond tiling work [2], [48]. Diamond tiling is an approach that we plan to apply to the shifted and fused schedules in the future.

Optimization in prior studies could be done in addition to the loop optimizations we investigate here. Kamil et al. [28] perform extensive auto tuning on a space of 4 levels of blocking for the spatial dimensions in the stencil and perform other optimizations like SIMDization. The Mint compiler [46] uses partial 3D blocking as described by Rivera and Tseng [38].

B. (Semi)-Automatic Parallelization

An auto parallelization approach [3], [18] can achieve good performance portably and frees the programmer from having to make fixed choices, but requires precise data dependence analysis that is difficult to perform for code written with libraries. Scripting transformation tools such as Chill [29], Orio [23], POET [49], and Sequoia mappings [17] enable performance programmers to specify transformations to apply to loops. We could almost use one of these tools to realize the schedules in this paper; however, we had to do the storage mapping by hand and we had to modify the code to perform overlapped tilling by hand.

Others have looked at the problem of determining the best temporary storage mapping given a schedule or vice versa [45], [14], [35], [41]. This is under the assumption that the schedule and storage mapping are affine, which is not the case for overlapped tile that need overlapped temporary storage.

VI. Experimental Results

The comparative efficiencies of the schedules were tested on two multi-core machines, described in detail below, that have performance levels typical of nodes found on current supercomputers. It is our goal to understand how node-level parallelism can be employed to allow efficient computation over increased box sizes above N = 16. As such, each schedule was tested for block sizes of N = 16, 32, 64, and 128. To test parallel scalability, each case was also executed over a range of thread counts ranging from one up to the maximum number of cores of the machine. We found the performance results for box sizes of N = 32 and 64 to fall smoothly in between those of N = 16 and 128 so we only report the data for the two extreme cases. Our primary result is that several of the schedules allow execution of the N = 128 sized boxes with an efficiency equal to that of the N = 16 case, paving the road for the move to larger box sizes. As tiling was a major building block of many of the variants, we also tested all tiled implementations with tile sizes of 4, 8, 16, and 32. We found that in general tile sizes of 8 and 16 were the most efficient.

For the tiled wavefront and overlapping tiles schedules, we tested two different means of parallelization. In the first approach, we parallelized over tiles within each box. In the second approach, we parallelized over boxes as ordinarily done in Chombo but iterated over the tiles serially within each box. Figure 9 shows the performance of the fastest schedules when parallelized over boxes (denoted "P>=Box") versus parallelized over tiles (denoted "P<Box"). The parallelization over boxes performs much better when the size of the boxes is small, but the two approaches have similar performance when the size of the box is large. The reason for this is simply that small boxes have too few tiles available per box to occupy the full number of threads. For example, a box of size N = 16, when implemented using tiles of size 16, only has one thread worth of tiles available per box, making the computation completely serial. As we are interested in the performance on large size boxes and because the performance is good for both approaches to parallelization in that case, we did not try to combine the two strategies to enable greater parallelism for small boxes.

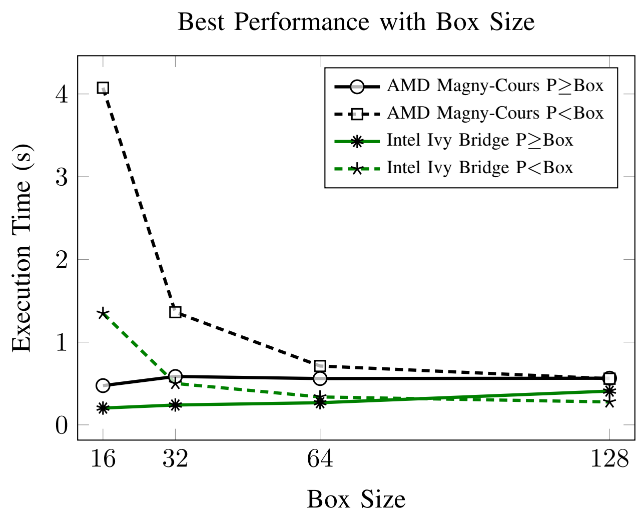


Fig. 9. Fastest performance over all configurations for each box size.

A. Experimental Setup

Each 24-core node on a Cray XT6M machine is composed of two AMD 12-core Magny Cours processors running at 1.90 GHz, configured with 32 GB of DDR3 RAM that deliver an aggregate memory bandwidth of 85.3 GB/s, shared between both sockets. Each core has a 64 KB of level 1 instruction cache, 64KB of level 2 data cache, and 512 KB of level 2 cache. All twelve cores on a socket share a 12 MB of level 3 cache.

Atlantis is a 20-core machine composed of two 10-core Intel Ivy Bridge E5-2670v2 chips running at a clock rate of 2.50 GHz. The system is configured with 128 GB of DDR3 RAM in a quad-channel configuration with a clock rate of 1600 MHz, giving 51.2 GB/s of bandwidth per socket or an aggregate system bandwidth of 102.4 GB/s. Each core has a 32 KB of level 1 instruction cache, 32 KB of level 1 data cache, and 256 KB level 2 cache. All cores on a socket share 25 MB of level 3 cache.

Each 16-core node on Cab uses two 8-core Intel Sandy Bridge E5-2670 chips running at a clock rate of 2.6 GHz.

Each node has 32 GB of DDR3 RAM configured in the same manner as Atlantis, giving the same per-socket and system bandwidth. The cache characteristics are also the same except that the level 3 cache is 20 MB in size.

In addition, a single-socket, 4-core Ivy Bridge desktop system was used to gather hardware counter data to aid in performance analysis of the implementations. The desktop machine allowed direct measurement of memory bandwidth data, which was not possible on the other machines due to lack of administrator level access. The system is equipped with an i5-3570K CPU running at 3.40 GHz and 16GB of DDR3 RAM running at a clock rate of 1458 MHz in a dual channel configuration, giving 21.0 GB/s of system bandwidth. The CPU has a 32 KB level 1 instruction cache, a 32 KB level 2 data cache, a 256 KB level 2 cache, and a 6MB level 3 cache, which is shared amongst all four cores. Bandwidth usage was measured using the Intel VTune Amplifier XE 2013 performance profiler tool and the Intel Performance Counter Monitor routines.

The test code was compiled on all machines using the GNU C++ compiler with OpenMP enabled. The optimization level was set to the -O3 level of optimizations, and loop unrolling and optimizations for the native architecture of each machine (-march=native command line parameter) were specifically enabled.

B. *Performance of the Schedules*

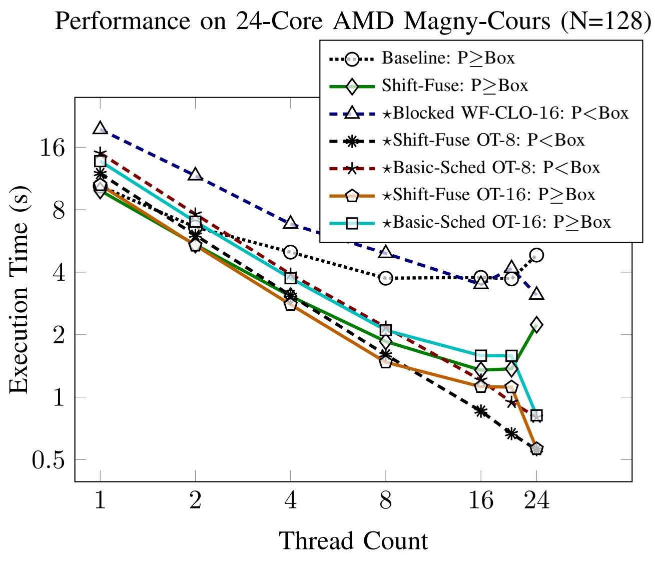


Fig. 10. Performance of various schedules when N=128. A ⋆ indicates the variant with a tile size giving the fastest performance was chosen. Note the good scalability of the wavefront schedule and the excellent scalability and performance of the overlapped tile schedules.

Series of modular loops. The performance of the original schedule is shown in Figures 2 through 4 for box sizes of N = 16 and N = 128 as the lines labeled "Baseline". The parallelism is over boxes so this case is essentially how Chombo code is executed today, except using OpenMP to parallelize over boxes instead of MPI. We can see that the scaling in the N = 16 case is basically perfect, but the scaling for the N = 128 case is terrible.

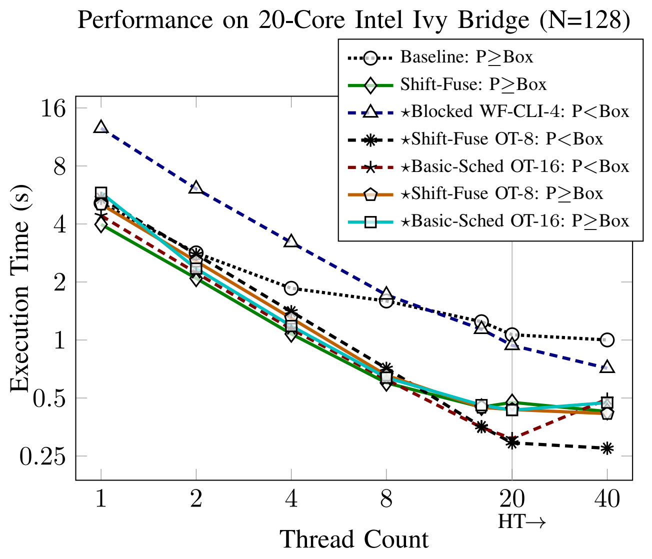


Fig. 11. Performance of various schedules when N=128. A ⋆ indicates the variant with a tile size giving the fastest performance was chosen. Note the good scalability of the wavefront schedule and the excellent scalability and performance of the overlapped tile schedules.

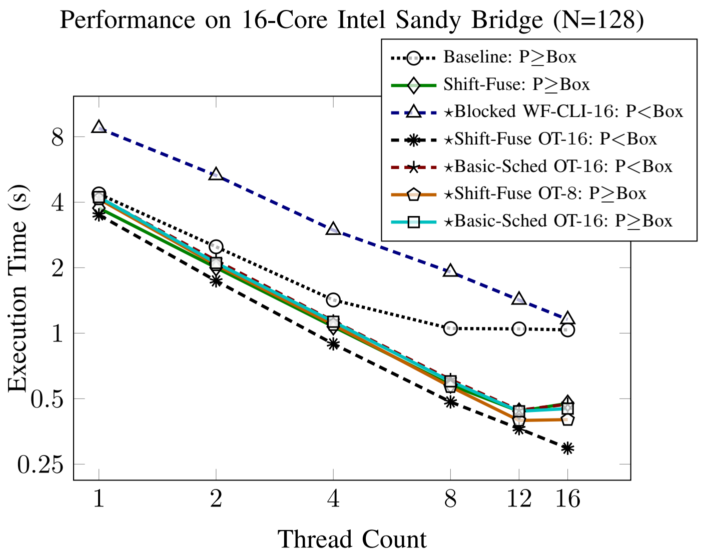


Fig. 12. Performance of various schedules when N=128. A ⋆ indicates the variant with a tile size giving the fastest performance was chosen. Note the good scalability of the wavefront schedule and the excellent scalability and performance of the overlapped tile schedules.

As discussed in Section IV, the amount of temporary data that must be stored and loaded is high. When the box size is N = 16, the data for the box, including temporaries, fits into the last level cache for all the machines. Since most loads and stores are absorbed by the cache, the memory bandwidth required is modest, and there is sufficient system bandwidth to allow scaling out to the full number of cores. As measured by VTune, the single-thread bandwidth profile when run on the Ivy Bridge desktop, while not constant throughout the entire execution, is composed of stretches of mostly sustained bandwidth up to 4.9 GB/s. When using the maximum of four threads for the system, the bandwidth increases to 14.5 GB/s which is well below the maximum system bandwidth. (Note that the bandwidth usage is not uniform, so the total cost at four cores is not necessarily four times the maximum cost of one core). The other machines have a larger number of cores but a correspondingly higher system bandwidth. We see from the excellent scalability shown in Figures 2 through 4 that the bandwidth has not been saturated on those machines for this schedule.

When N = 128, the working set can no longer be held within the last level of cache, and the bandwidth usage increases. On the Ivy Bridge desktop, the kernel exhibited bandwidth usage up to 18.3 GB/s for a single thread. The system bandwidth of 21.0 GB/s became highly contended when using two or more threads and the performance ceased to improve at all beyond two threads. For the HPC-class machines, we see in Figures 2 through 4 that their scalability falters after only a few threads, assumedly because their bandwidth has become contended as well.

We conclude that a major key to improving scalability on large boxes, compared to the base case of N = 16, is to reduce the bandwidth usage of the schedules. In the highest performing schedules, this is achieved through a combination of loop fusion to reduce the number of temporaries and tiling to allow more memory accesses to be absorbed by the cache.

Shifted and Fused Loop. Shifting and fusing loops significantly lowers the number of temporaries that must be stored and subsequently loaded. This has the effect of lowering the bandwidth required, as a greater proportion of the cycles are spent performing computations rather than loads and stores. On the Ivy Bridge desktop, for the N = 16 case, the single-thread bandwidth is lowered from 4.9 GB/s in the un-fused case to 3.9 GB/s. The change is small due to the good caching behavior of the baseline at that size. However, on the N = 128 case, the bandwidth usage reduces to time stretches requiring 9.4 GB/s interleaved with time intervals of similar length requiring less than 6 GB/s, which is a significant reduction in bandwidth cost compared to the 18.3 GB/s for the baseline case. This allows the schedule to scale to a larger number of threads before there is significant contention for memory bandwidth.

In Figures 10 through 12, the shifted and fused kernel when parallelized over boxes is denoted “Shift-Fuse: P>=Box”. The graphs show that the scalability of that case is greatly improved on the HPC machines compared to the baseline. For both machines, the schedule exhibits nearly perfect scaling up to 8 threads, a significant increase compared to the non-fused case. However, we also see that shifting and fusing alone is inadequate to achieve ideal scalability over all cores of the machine.

Wavefront Tiling. All of the wavefront schedules were implemented with shifting and fusing as well as tiling. In the figures, this is denoted as “Blocked Wavefront”. As a result, the bandwidth cost of the schedules is low especially at tile sizes of 16 or smaller, where the data still fits into last level cache. Therefore, on boxes of size N = 128, the wavefront implementations, except those with tiles of size 32, generally had good scaling out to the full number of cores for the system. In Figures 10 through 12, the wavefront variants that performed best on each particular machine are denoted “Blocked WF-CLO-16: P<Box” and “Blocked WF-CLI-4: P<Box” respectively. Recall that “P<Box” stands for “parallelized over tiles”. We see that the scaling for those schedules is good but that the lines for the wavefront schedules are offset above the other lines on the graph. In other words, the schedules scaled well but still had a high time cost compared to the other schedules. This is because the wavefront schedules provide only limited parallelism. During the first several wavefronts, there are not enough tiles available to keep every core busy, resulting in an initial warm-up period with low parallelism (see Figure 8(a)). This results in a higher overall time cost compared with schedules that can make full use of their threads throughout the computation, as in the overlapped tiling case below. Overall, wavefront schedules were not competitive.

Overlapped Tiling. Overlapped tiling schedules exhibited the best performance of all the schedules for computing large boxes. They combine the low bandwidth cost of tiling with maximum parallelism, as each thread can be assigned to a completely independent tile. Figures 10 through 12 show the fastest performing overlapped tile schedules when using simple tiling and also when adding shift-and-fuse in two cases each: when parallelized over boxes and when parallelized over tiles. The overlapped tiles schedules are labeled “OT”, and are the last four schedules in the legend of each graph. The shifted-and-fuse variant is labeled “Shift-Fuse” and the simple tiled version is written “Basic-Sched”. The number after the “OT-” is the tile size in each case. Recall that “P>=Box” means parallelized over boxes while “P<Box” denotes parallelization over tiles. Overall, we see these schedules exhibit excellent scalability and greatly improved performance over the baseline case.

Figures 2 through 4 show that the efficiency of the fastest performing overlapped tiling schedules, when applied to boxes of size N = 128, is comparable to that of the baseline schedule when applied to boxes of size N = 16. The optimization greatly outperforms the baseline schedule applied to the N = 128 box size. This is the primary result of this paper: demonstration that a sufficiently well-implemented schedule can achieve the performance efficiency enjoyed by computing over small boxes while avoiding their high ghost-cell exchange penalty.

VII. Conclusions

Using larger box sizes in PDE codes would enable reducing ghost cell overhead, but straight-forward parallelization over boxes or over cells within boxes using large box sizes results in poor parallel scaling on multicore architectures due to memory bandwidth bottlenecks. To solve this problem, we study around 30 different inter-loop scheduling strategies for a CFD benchmark. Our results show that, for an AMD Magny-Cours (Cray XT6m node), an Ivy Bridge node, and a Sandy Bridge node, scheduling variants that optimize for data locality and minimize temporary storage can provide for large box sizes 128^3 the same performance and/or parallel scalability as for smaller box sizes 16^3. In some cases, the performance of the small box sizes improve as well. This result suggests that it would be beneficial to determine ways to automate the automatic implementation, selection, and tuning of such inter-loop program optimizations for PDE application frameworks.

VIII. Acknowledgments

We would like to thank Brian Van Straalen for discussions about this work and answering questions about Chombo. Support for this work was provided through the Resilient Extreme-Scale Solvers program funded by the U.S. Department of Energy, Office of Science, Advanced Scientific Computing Research. This work was also supported by the Department of Energy CACHE Institute grant DE-SC04030 and DOE grant DE-SC0003956. This research utilized the CSU ISTeC Cray HPC System supported by NSF Grant CNS-0923386. This work was performed under the auspices of the U.S. Department of Energy (DOE) by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344.

References

[1] G. Allen, T. Dramlitsch, I. Foster, T. Goodale, N. Karonis, M. Ripeanu, E. Seidel, and B. Toonen. Cactus-g toolkit: Supporting efficient execution in heterogeneous distributed computing environments. In *In Proceedings of 4th Globus Retreat*, 2000.

[2] V. Bandishti, I. Pananilath, and U. Bondhugula. Tiling stencil computations to maximize parallelism. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis (SC)*, 2012.

[3] U. Banerjee, R. Eigenmann, A. Nicolau, and D. A. Padua. Automatic program parallelization. *Proceedings of the IEEE*, 81(2):211–243, 1993.

[4] F. Bassetti, K. Davis, and D. Quinlan. Optimizing transformations of stencil operations for parallel object-oriented scientific frameworks on cache-based architectures. *Lecture Notes in Computer Science*, 1505, 1998.

[5] P. Basu, S. W. Williams, B. V. Straalen, A. Venkat, L. Oliker, and M. Hall. Compiler generation and autotuning of communication-avoiding operators for geometric multigrid. In *IEEE Conference on High Performance Computing*, December 18–21 2013.

[6] M. Berger and P. Colella. Local adaptive mesh refinement for shock hydrodynamics. *J. Comput. Phys.*, 82(1):64–84, 1989.

[7] M. J. Berger and J. Oliger. Adaptive mesh refinement for hyperbolic partial differential equations. *J. Comput. Phys.*, 53(3):484–512, 1984.

[8] C. Chen. Polyhedra scanning revisited. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2012.

[9] C. Chen, M. Hall, and A. Venkat. Omega+. http://ctop.cs.utah.edu/ctop/?page_id=21, August 15 2012.

[10] M. Christen, O. Schenk, and H. Burkhart. Patus: A code generation and autotuning framework for parallel iterative stencil computations on modern microarchitectures. In *Proceedings of the IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2011.

[11] M. Christen, O. Schenk, E. Neufeld, M. Paulides, and H. Burkhart. Manycore Stencil Computations in Hyperthermia Applications. In J. Dongarra, D. Bader, and J. Kurzak, editors, *Scientific Computing with Multicore and Accelerators*, pages 255–277. CRC Press, 2010.

[12] P. Colella, D. T. Graves, N. Keen, T. J. Ligocki, D. F. Martin, P. McCorquodale, D. Modiano, P. Schwartz, T. Sternberg, , and B. V. Straalen. Chombo software package for amr applications - design document. Technical report, Lawrence Berkeley National Laboratory, 2009.

[13] J. Cong and Y. Zou. Fpga-based hardware acceleration of lithographic aerial image simulation. *ACM Trans. Reconfigurable Technol. Syst.*, 2(3):17:1–17:29, Sept. 2009.

[14] A. Darte, R. Schreiber, and G. Villard. Lattice-based memory allocation. *IEEE Transactions on Computers*, 54(10):1242–1257, 2005.

[15] R. Deiterding. Detonation structure simulation with amroc. In L. Yang, O. Rana, B. Martino, and J. Dongarra, editors, *High Performance Computing and Communications*, volume 3726 of *Lecture Notes in Computer Science*, pages 916–927. Springer Berlin Heidelberg, 2005.

[16] C. Ding and Y. He. A ghost cell expansion method for reducing communications in solving pde problems. In *Proceedings of the ACM/IEEE Conference on Supercomputing*, Supercomputing '01, pages 50–50, New York, NY, USA, 2001. ACM.

[17] K. Fatahalian, T. J. Knight, M. Houston, M. Erez, D. R. Horn, L. Leem, J. Y. Park, M. Ren, A. Aiken, W. J. Dally, and P. Hanrahan. Sequoia: Programming the memory hierarchy. In *Proceedings of the 2006 ACM/IEEE Conference on Supercomputing*, 2006.

[18] P. Feautrier. Automatic parallelization in the polytope model. In *The Data Parallel Programming Model*, pages 79–103, 1996.

[19] C. for Computational Sciences and L. B. N. L. Engineering. Boxlib, Sept. 2012.

[20] T. Goodale, G. Allen, G. Lanfermann, J. Massó, T. Radke, E. Seidel, and J. Shalf. The Cactus framework and toolkit: Design and applications. In *Vector and Parallel Processing – VECPAR'2002, 5th International Conference, Lecture Notes in Computer Science*, Berlin, 2003. Springer.

[21] J. Guo, G. Bikshandi, B. B. Fraguela, M. J. Garzaran, and D. Padua. Programming with tiles. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pages 111–122, New York, NY, USA, 2008. ACM.

[22] S. M. J. Guzik, P. McCorquodale, and P. Colella. A freestream-preserving high-order finite-volume method for mapped grids with adaptive-mesh refinement. AIAA 2012-0574, 50th AIAA Aerospace Sciences Meeting, 2012.

[23] A. Hartono, B. Norris, and S. Ponnuswamy. Annotation-based empirical performance tuning using Orio. In *23rd IEEE International Parallel & Distributed Processing Symposium (IPDPS) Rome, Italy*, May 2009.

[24] W. Henshaw. Overture: An object-oriented toolkit for solving partial differential equations in complex geometry, Mar. 2014.

[25] J. Holewinski, L. Pouchet, and P. Sadayappan. High-performance code generation for stencil computations on gpu architectures. In *Proceedings of the 26th ACM international conference on Supercomputing*, pages 311–320. ACM, 2012.

[26] R. D. Hornung and S. R. Kohn. Managing application complexity in the samrai object-oriented framework. *Concurrency and Computation: Practice and Experienc*, 14:347–368, 2002.

[27] Q. Huang and J. Xue. Code tiling for improving the cache performance of pde solvers. In *In Proceedings of the International Conference on Parallel Processing. ACM*, pages 615–625, 2003.

[28] S. Kamil, C. Chan, L. Oliker, J. Shalf, and S. Williams. An auto-tuning framework for parallel multicore stencil computations. In *Parallel Distributed Processing (IPDPS), 2010 IEEE International Symposium on*, pages 1–12, 2010.

[29] M. Khan, P. Basu, G. Rudy, M. Hall, C. Chen, and J. Chame. A script-based autotuning compiler system to generate high-performance cuda code. *ACM Trans. Archit. Code Optim.*, 9(4):31:1–31:25, Jan. 2013.

[30] S. Krishnamoorthy, M. Baskaran, U. Bondhugula, J. Ramanujam, A. Rountev, and P. Sadayappan. Effective automatic parallelization of stencil computations. In *Proceedings of Programming Languages Design and Implementation (PLDI)*, volume 42, pages 235–244, New York, NY, USA, 2007. ACM.

[31] J. Luitjens, B. Worthen, M. Berzins, and T. Henderson. Scalable parallel AMR for the Uintah multiphysics code. In D. A. Bader, editor, *Petascale Computing Algorithms and Applications*. Chapman and Hall/CRC, Boca Raton, FL, 2008.

[32] P. McCorquodale and P. Colella. A high-order finite-volume method for conservation laws on locally refined grids. *Comm. App. Math. Comput. Sci.*, 6(1):1–25, 2011.

[33] J. Meng and K. Skadron. Performance modeling and automatic ghost zone optimization for iterative stencil loops on gpus. In *Proceedings of the 23rd international conference on Supercomputing*, ICS '09, pages 256–265, New York, NY, USA, 2009. ACM.

[34] A. Nguyen, N. Satish, J. Chhugani, C. Kim, and P. Dubey. 3.5-d blocking optimization for stencil computations on modern cpus and gpus. In *Proceedings of the ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '10, pages 1–13, Washington, DC, USA, 2010. IEEE Computer Society.

[35] F. Quilleré and S. Rajopadhye. Optimizing memory usage in the polyhedral model. *ACM Transactions on Programming Languages and Systems*, 22(5):773–815, 2000.

[36] J. Ragan-Kelley, C. Barnes, A. Adams, S. Paris, F. Durand, and S. Amarasinghe. Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. In *Proceedings of the 34th ACM SIGPLAN conference on Programming language design and implementation*, PLDI '13, pages 519–530, New York, NY, USA, 2013. ACM.

[37] F. Rastello and T. Dauxois. Efficient tiling for an ode discrete integration program: Redundant tasks instead of trapezoidal shaped-tiles. In *Proceedings of the 16th International Parallel and Distributed Processing Symposium*, IPDPS '02, pages 138–, Washington, DC, USA, 2002. IEEE Computer Society.

[38] G. Rivera and C.-W. Tseng. Tiling optimizations for 3d scientific computations. In *Supercomputing*, 2000.

[39] G. Roth, J. Mellor-Crummey, K. Kennedy, and R. G. Brickner. Compiling stencils in high performance fortran. In *Proceedings of the ACM/IEEE Conference on Supercomputing (CDROM)*, pages 1–20, New York, NY, USA, 1997. ACM Press.

[40] A. Sawdey and M. T. O'Keefe. Program analysis of overlap area usage in self-similar parallel programs. In *Proceedings of the 10th International Workshop on Languages and Compilers for Parallel Computing*, LCPC '97, pages 79–93, London, UK, UK, 1998. Springer-Verlag.

[41] M. M. Strout, L. Carter, J. Ferrante, and B. Simon. Schedule-independent storage mapping for loops. In *Proceedings of the Eighth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 24–33, San Jose, California, October 3–7, 1998.

[42] Y. Tang, R. A. Chowdhury, B. C. Kuszmaul, C.-K. Luk, and C. E. Leiserson. The pochoir stencil compiler. In *Proceedings of the 23rd ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '11, pages 117–128, New York, NY, USA, 2011. ACM.

[43] T. C. D. Team. Clawpack-5, 2013.

[44] R. U. The Applied Software Systems Laboratory. Grace: Grid adaptive computational engine, Mar. 2012.

[45] W. Thies, F. Vivien, and S. Amarasinghe. A step towards unifying schedule and storage optimization. *ACM Transactions on Programming Languages and Systems*, 29(6):34, 2007.

[46] D. Unat, X. Cai, and S. B. Baden. Mint: realizing cuda performance in 3d stencil methods with annotated c. In *Proceedings of the international conference on Supercomputing*, ICS '11, pages 214–224, New York, NY, USA, 2011. ACM.

[47] D. Wonnacott. Using time skewing to eliminate idle time due to memory bandwidth and network limitations. In *Proceedings of the 14th International Symposium on Parallel and Distributed Processing*, IPDPS '00, Washington, DC, USA, 2000. IEEE Computer Society.

[48] D. G. Wonnacott and M. M. Strout. On the scalability of loop tiling techniques. In *Proceedings of the 3rd International Workshop on Polyhedral Compilation Techniques (IMPACT)*, January 2013.

[49] Q. Yi, K. Seymour, H. You, R. Vuduc, and D. Quinlan. Poet: Parameterized optimizations for empirical tuning. In *Proceedings of the Parallel and Distributed Processing Symposium*, 2007.

[50] X. Zhou, J.-P. Giacalone, M. J. Garzarán, R. H. Kuhn, Y. Ni, and D. Padua. Hierarchical overlapped tiling. In *Proceedings of the Tenth International Symposium on Code Generation and Optimization*, CGO '12, pages 207–218, New York, NY, USA, 2012. ACM.